

**2020** 级

《大数据存储系统与管理》课程

**实 验 报 告**

**姓 名 付景洋**

**学 号 U202015424**

**班 号 计算机2005班**

**日 期 2023.5.25**

**目 录**

[一、实验目的 1](#_Toc135935991)

[二、实验背景 1](#_Toc135935992)

[三、实验环境 1](#_Toc135935993)

[四、实验内容 2](#_Toc135935994)

[4.1对象存储技术实践 2](#_Toc135935995)

[4.2 对象存储性能分析 2](#_Toc135935996)

[五、实验过程 3](#_Toc135935997)

[5.1对象存储系统搭建 3](#_Toc135935998)

[5.2对象存储服务使用 3](#_Toc135935999)

[5.3对象存储性能分析 5](#_Toc135936000)

[六、实验总结 7](#_Toc135936001)

[参考文献. 8](#_Toc135936002)

# 一、实验目的

1. 熟悉对象存储技术，代表性系统及其特性；

2. 实践对象存储系统，部署实验环境，进行初步测试；

3. 基于对象存储系统，分析性能问题，架设应用实践。

# 二、实验背景

对象存储是一种将数据集合作为单个、唯一可识别的不可分割项或对象存储和检索的方法。它适用于可以封装和管理为对象的任何形式的数据。

对象被视为一个原子单元。与文件系统中的目录层次结构相对应的结构不存在；每个对象在系统中通过唯一的对象标识符进行唯一标识。

在这种存储类型上创建对象时，整个数据集合都被处理和处理，而不考虑它可能具有的子部分。在从对象存储中读取时，可以读取整个对象，或者要求读取其部分。通常无法更新对象或对象的部分，需要重新编写整个对象。大多数对象存储允许删除对象。

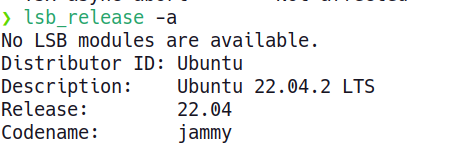
存储系统不关心对象包含的内容。它们可以是简单的数据集合、文件、整个文件系统、视频、虚拟机或容器、数据库等等，因为存储系统只将对象视为与一块数据相关联的对象标识符。

对象存储如何存储对应于应用程序并不重要。许多系统使用基于文件或块的存储；有些系统专门设计了直接支持对象的设备。无论如何存储，所有对象存储系统都支持简单的API来创建、读取和删除对象，并搜索元数据。其中一些较为知名的是SNIA的云数据管理接口（CDMI）和Amazon的S3，它们通过HTTP协议进行操作使得对象存储对Web和云的使用具有吸引力。

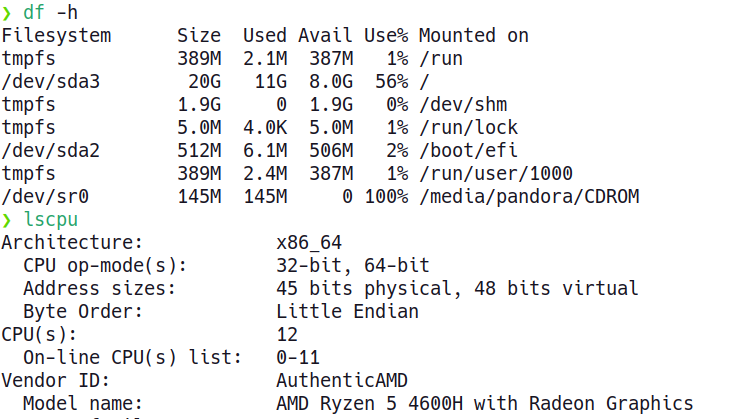
OpenStack Object Store项目，也被称为Swift，提供了云存储软件，可以通过简单的API存储和检索大量数据。它专为大规模操作而构建，并在整个数据集上进行了耐久性、可用性和并发性的优化。Swift非常适合存储可以无限增长的非结构化数据。

# 三、实验环境

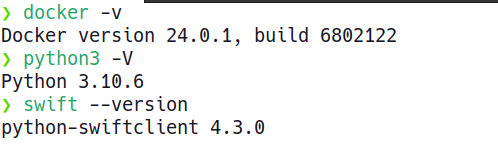
本实验在个人电脑采用VMware上Ubuntu虚拟机安装docker搭建对象存储系统。



为虚拟机分配了12个内核和20G的存储空间。



安装了docker、python-swiftclient等相关软件。



# 四、实验内容

## 4.1对象存储技术实践

根据个人情况选择适合自己的对象存储实验系统并完成对象存储系统搭建，并体验对象存储系统的功能服务。使用客户端发送 PUT 、GET、DELETE 请求，在服务端终端以及本地文件夹下观测服务端的响应结果。

本实验选用OpenStack Swift搭建对象存储系统，使用[Docker OpenStack Swift onlyone](https://hub.docker.com/r/fnndsc/docker-swift-onlyone)简易容器部署，使用[python-swiftclient](https://github.com/openstack/python-swiftclient)提供的Swift API编写python程序进行标准测试，通过测试吞吐率、延迟等指标，分析对象尺寸、并发率等因素对性能的影响。

## 4.2 对象存储性能分析

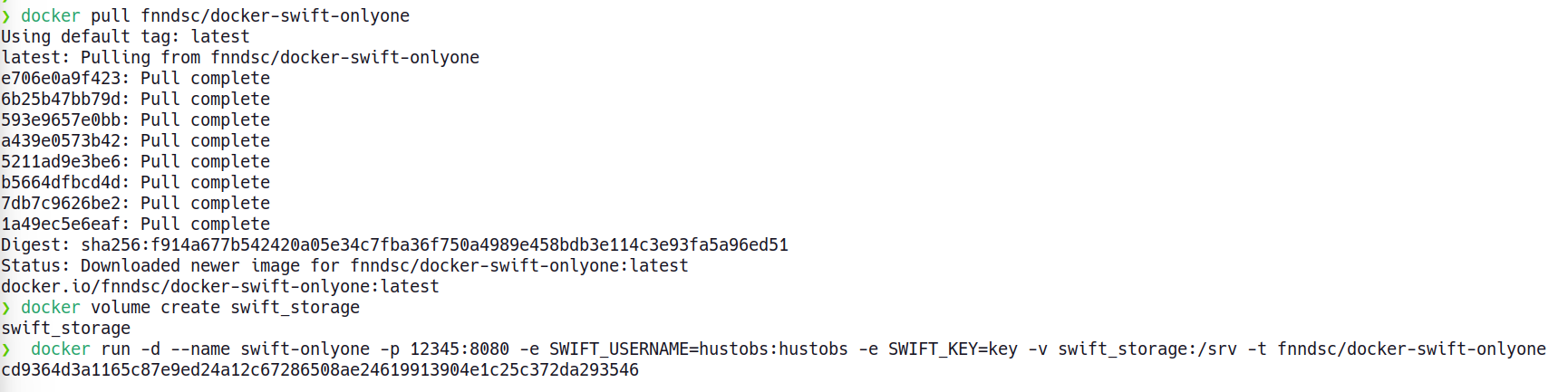
在进行性能观测中容易发现对象存储系统存在尾延迟问题，通过绘制[eCDF](https://en.wikipedia.org/wiki/Empirical_distribution_function)量化观察尾延迟现象，思考解决尾延迟问题的方法。

解决尾延迟问题主要有对冲请求和关联请求两种方案。延迟高的情况为小概率事件，一个简单的想法是客户端同时对多个服务器发送请求，接受第一个响应便能多个延迟取最小，然而这种方法会使负载大大增加，对冲请求就是该方法的一个变种，它在第一个请求的响应时间超过95%的延迟时才发送第二个请求，减少额外负载的同时显著缩短尾延迟。使用更激进的对冲请求方案(95%的比例改为更小)则会消耗更多的资源，当请求在某个服务器开始执行时，及时关闭其他服务器上的该请求有助于减少资源消耗。关联请求的方法就是将服务器相互关联，客户端同时向多个服务器发送请求，服务器收到的请求中包含其他服务器的信息，当第一个服务器开始执行时就能及时向其他服务器发送关闭该请求的信息。

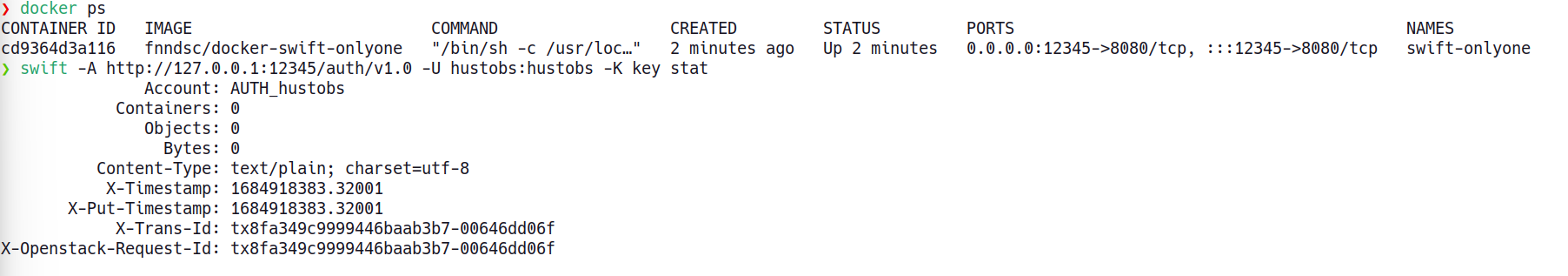
# 五、实验过程

## 5.1对象存储系统搭建

拉取fnndsc/docker-swift-onlyone镜像，准备docker数据卷，创建swift-onlyone容器并挂载到其中，并设置用户名和密码为hustobs。



查看容器创建成功。

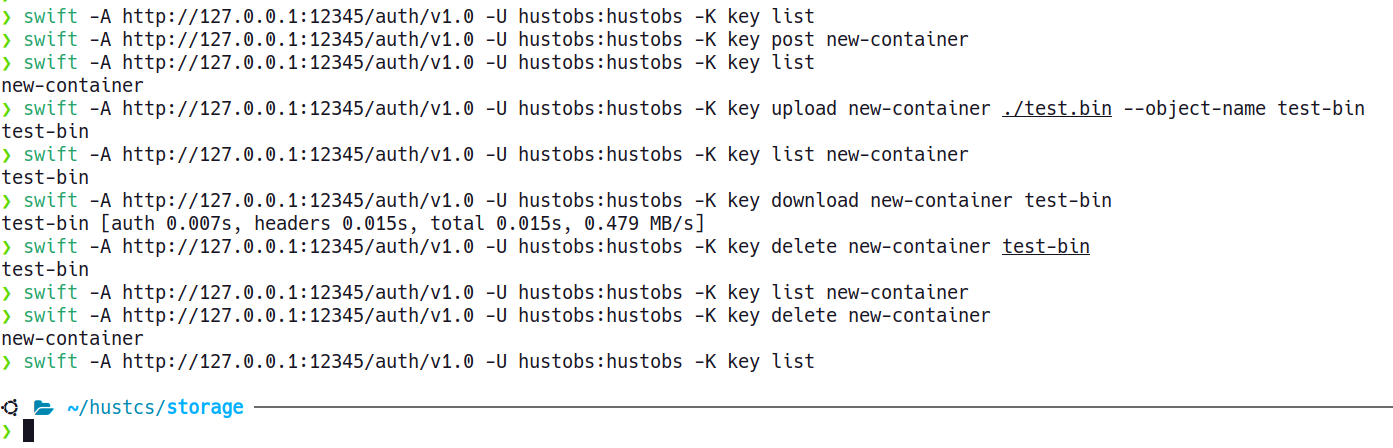


## 5.2对象存储服务使用

在计算机领域中，create, read, update, and delete (缩写为 CRUD) 是访问持久存储的4项基本操作，在HTTP中体现为PUT、GET、DELETE三种核心操作。

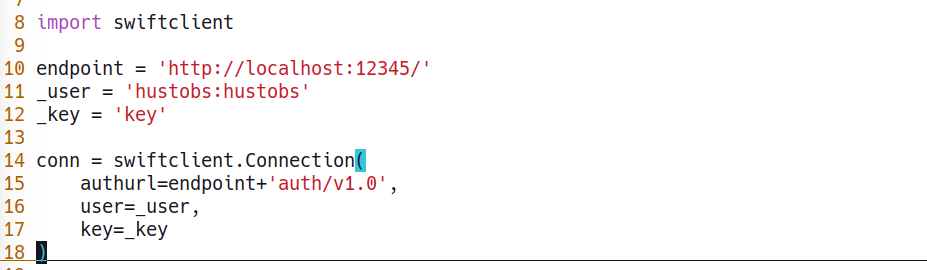
python-swiftclient同时提供了[CLI](https://docs.openstack.org/python-swiftclient/latest/cli/index.html)和[API](https://docs.openstack.org/python-swiftclient/latest/client-api.html)。

验证Swift CLI的功能，首先创建新容器new-container，向容器中上传大小为4K的本地文件test.bin，将其对象命名为test-bin，再从容器中下载该对象，最后分别删除对象和容器。

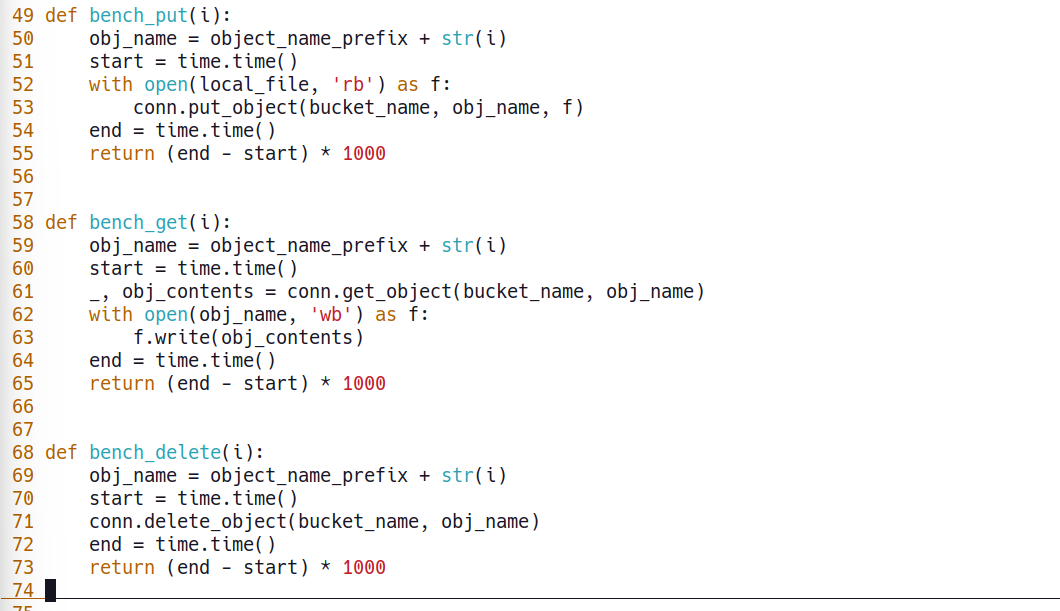


实际上在进行存储性能分析时采用Connection API，其中并发请求部分参考了obs-tutorial中的[latency-collect.ipynb](https://github.com/ShiZhan/obs-tutorial/blob/master/latency-collect.ipynb)，通过设置不同的客户端线程数量、对象尺寸和对象数量测试吞吐率、延迟等指标。

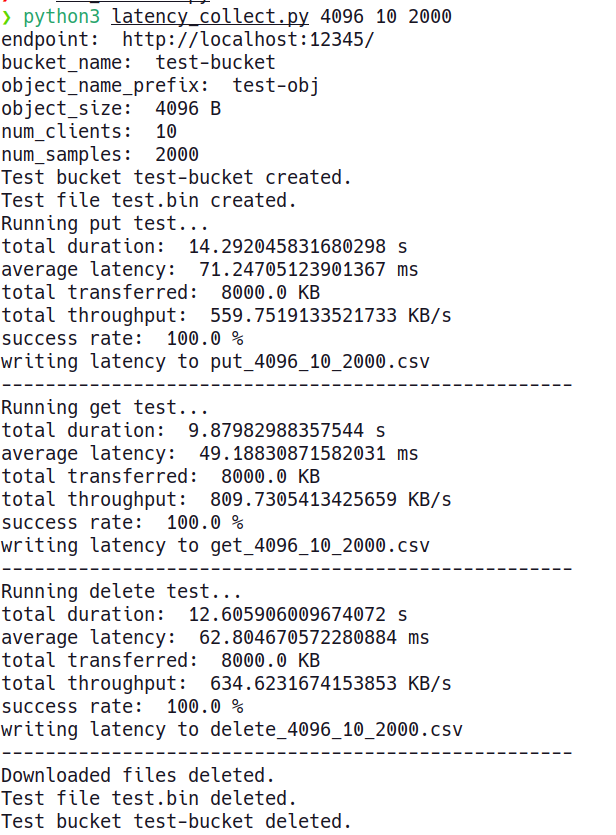
客户端进行连接的代码如下图所示。



调用Swift API接口PUT、GET、DELETE的代码如下图所示。

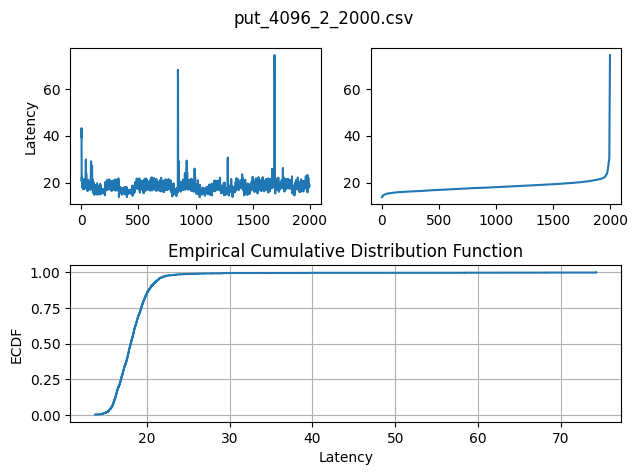


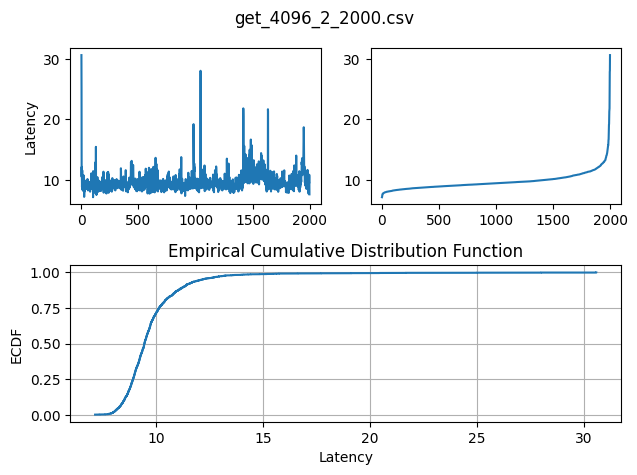
输入参数object\_size、num\_clients、num\_samples分别为4096B、10、2000运行如下图所示。

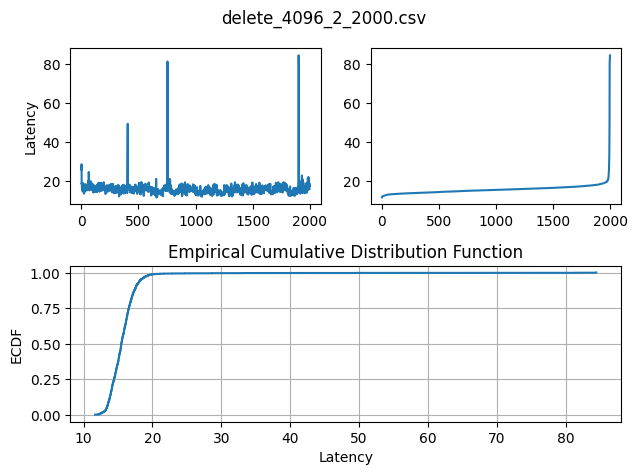


## 5.3对象存储性能分析

以object\_size=4096B，num\_clients=2，num\_samples=2000为例，绘制图表如下所示，程序见latency\_plot.ipynb。

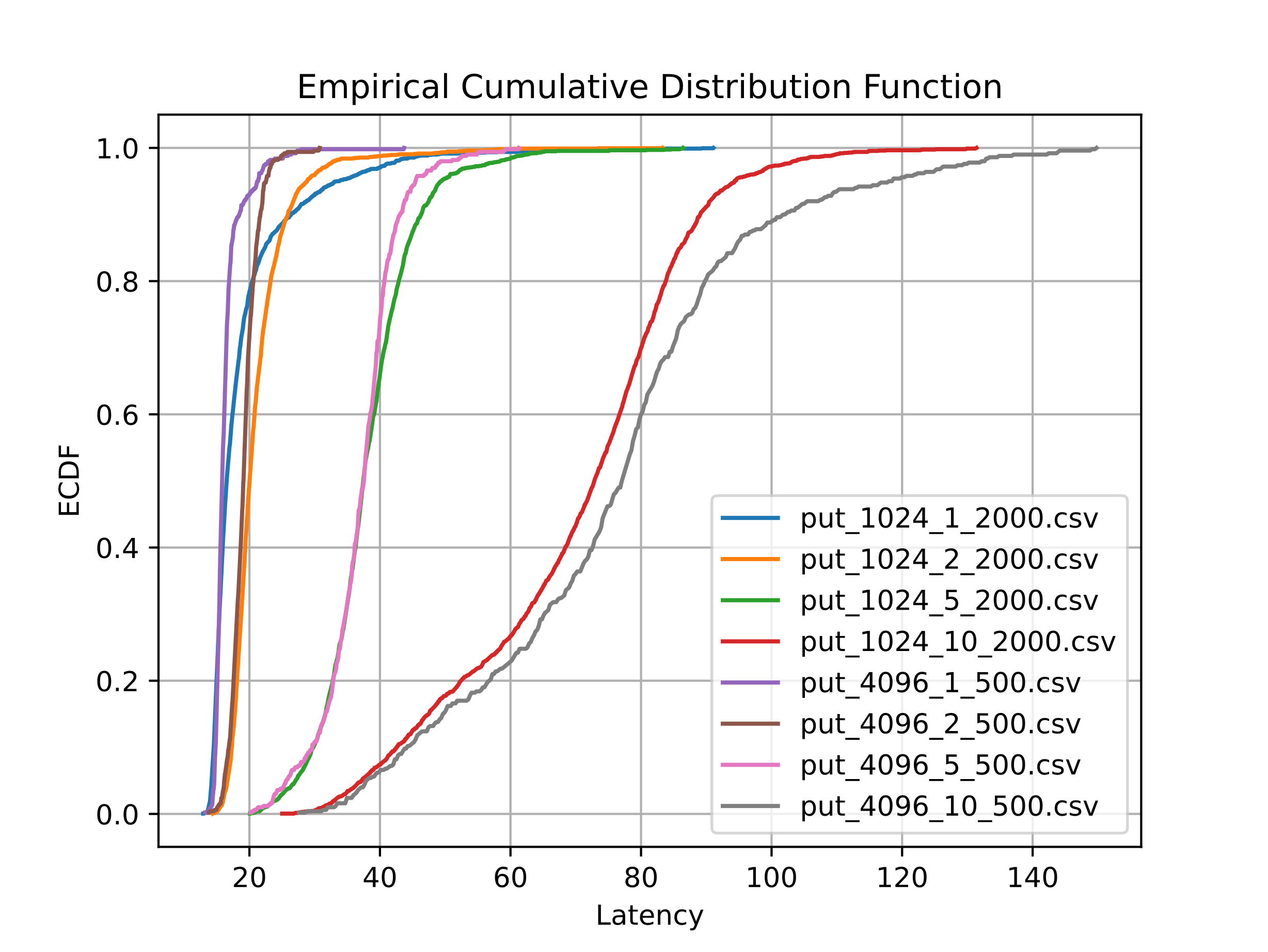






可以观察到大部分请求的响应延迟较小，而少部分请求响应延迟较大，即为尾延迟现象。另外get、delete的延迟明显高于put的延迟，可能是因为前者在向对象系统写入、删除数据的同时产生了一些副作用，而后者只需要下载数据，[这也与一般HTTP请求的特点相同](https://stackoverflow.com/questions/1211881/why-is-the-get-method-faster-than-post-in-http)。

探究对象尺寸和并发数对性能的影响，将num\_samples和object\_size的固定，选取put操作，改变object\_size和num\_clients的大小，分析实验数据可知增大对象尺寸能够有效增大吞吐率，并发数增加也能减小总延迟，但是当并发数增大到一定程度时平均延迟增加，可能是线程之间发生阻塞的原因。



# 六、实验总结

本次实验为我打开了对象存储技术的新大门，我初步接触了解了对象存储技术的特点和相关经典架构，并对搭建简易的对象存储系统服务端和客户端进行了实践，并对对象存储系统的性能进行了观测，对对象存储系统性能的影响因素有了更深刻的认识，本次实验提升了我设计实验分析数据的能力。

实验的很多地方都值得我进一步学习研究，为解决尾延迟问题，我还阅读了相关文献，对缓解尾延迟现象的方法有了一定的了解，但受限于时间和自身的知识储备，很遗憾没有编程实现相关算法。

# 参考文献.

[1] ARNOLD J. OpenStack Swift[M]. O’Reilly Media, 2014.

[2] Dean J, Barroso L A. Association for Computing Machinery, 2013. The Tail at Scale[J]. Commun. ACM, 2013, 56(2): 74–80.